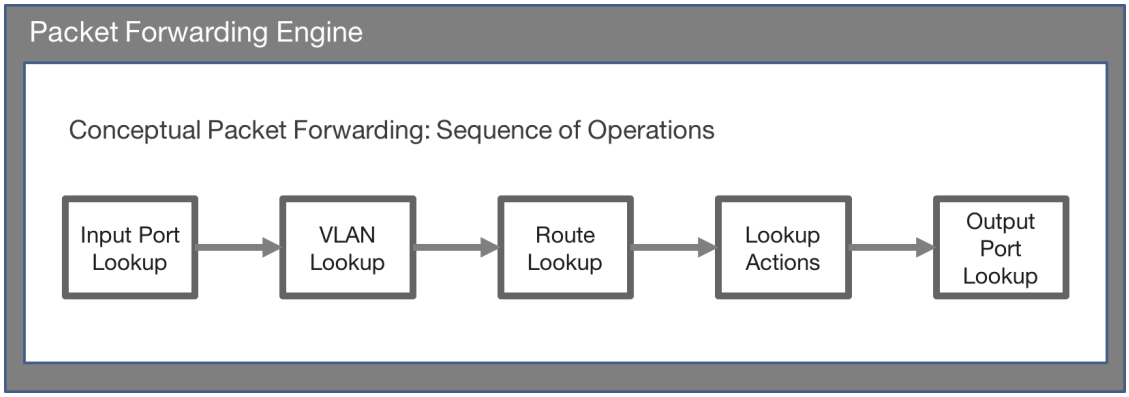
Advance Forwarding Interface (AFI)

Table of Contents

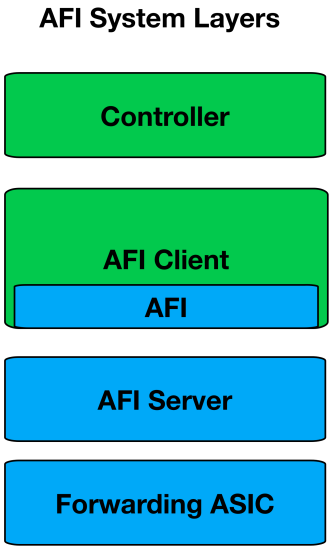
1. Introduction

Packet forwarding, at its core, is a sequence of operations executed by a packet forwarding engine (PFE). The packet forwarding engine, when given an input packet, performs this sequence of operations on the packet.



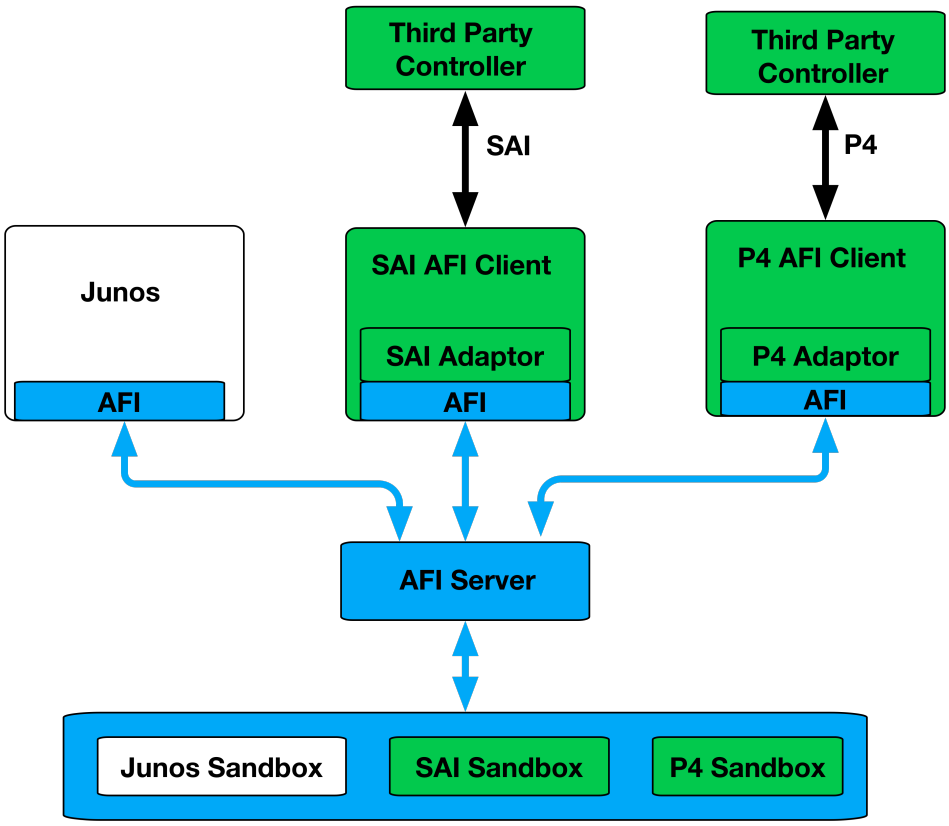
Each of these operations can be represented by a node in a graph of potential packet forwarding operations. When packet is given to packet forwarding engine, it executes the operations in the graph that match the packet.

Advanced Forwarding Interface (AFI) provides third party developers with ability to control and manage a section of forwarding path graph.



Third-party developers, who would typically use AFI, include:

- Non-PFE developers inside Juniper (for example the ASIC team writing/automating their test cases using AFI)
- Developers outside Juniper (Juniper customers) writing AFI client applications – who would create/control a forwarding sandbox in forwarding chip to achieve a specific forwarding path functionality such as Lightweight 4over6, SAI or P4



2. Advanced Forwarding Interface (AFI)

AFI provides clients with the ability to program a section of the forwarding path via a small virtual container called a **sandbox**. Sandboxes are allocated and provided by the underlying system (through CLI configuration). Once allocated, sandboxes are connected to the underlying forwarding engine or other sandboxes by their input and output ports. Sandboxes are assigned simple textual names that allow them to be identified by clients and the server.

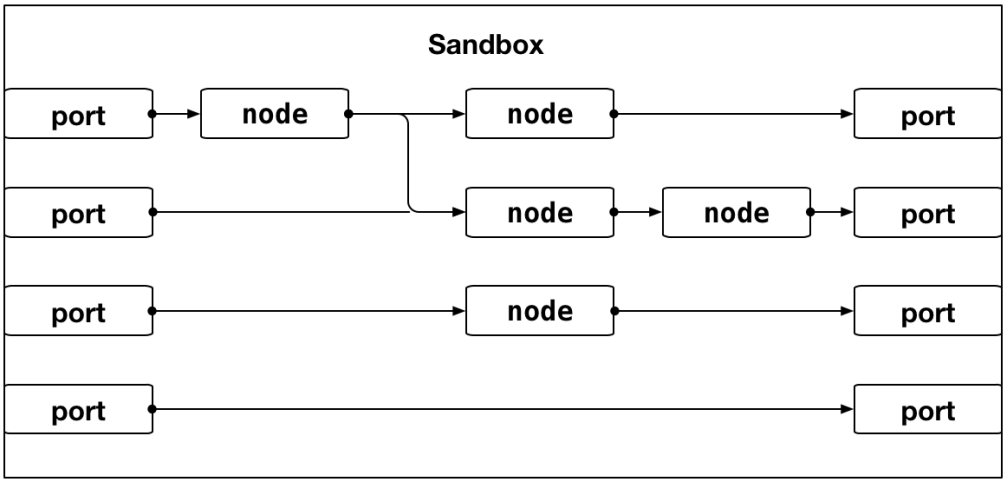
AFI system comprises of an AFI client and server. The client and the server run as different processes. The connection between the client and server is provided and managed by the AFI transport. An AFI client is a translation layer between higher-level constructs (route tables, routes, next-hops) and AFI nodes (lookup tables, packet field matches and packet rewrites).

All the connection and interface semantics are abstracted into the AFI transport. All the state that the AFI server requires to program the forwarding path is supplied via the transport.

Terminology

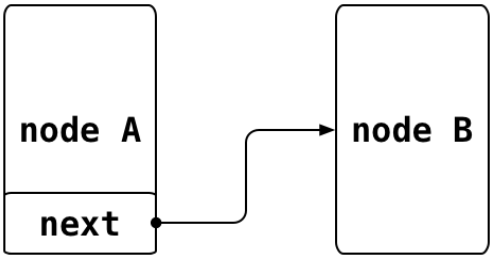
Sandbox	A sandbox is essentially a walled area within forwarding path. The part of forwarding node graph inside the sandbox is controlled/managed by an AFI client.
Port	Ports are the points where network lookup jumps into and out of the sandbox. Each sandbox has a set of ports associated with it.
Node	Basic forwarding path elements in AFI are called nodes. Different nodes may vary in their complexity and connection options. Example of nodes are tables, trees, lookups and conditionals.
Token	All nodes have a 64-bit token assigned to them that uniquely identifies them within the sandbox that owns them. Each sandbox manages its own token space.
Entry	Entries are the individual matches in a lookup container (such as tree, table, hash etc.). Entries are the individual match elements for a container. Entry is, at its core, a tuple of three values; a token value that describes the container the entry exists in (e.g. a tree), a unique key value for the entry (which is variable length) and a token which points at what to do next when we perform a lookup and match on the entry's key value.
Field	Fields are the basic descriptors that refer to any variable used by a node. Each field is defined by a string. For example field refers to a field in a packet (e.g. "packet.ip4.ttl", "packet.ip6.daddr" or "packet.mpls.label"). Fields are also used to refer to any metadata used during a lookup (e.g. "meta.ifinput", "meta.nhid" or "meta.ifoutput").
Data	AFI provides a standard method for defining data to be used by the AFI. The data classes provide support for scalars, IPv4/IPv6/Ethernet addresses, abstract prefixes and other commonly used types.
Key	The tuple of a field and data is called a key. AFI uses keys (specific field and data pairs) for describing an individual match condition.

The basic container for an AFI client is a **sandbox**. A sandbox offers the client a view of a section of the forwarding path. Each client is assigned a sandbox to control and build into. Sandboxes are assigned system-unique names that allow them to be identified by clients. The sandbox is a container, into which a client can insert and connect the forwarding path nodes that it requires.



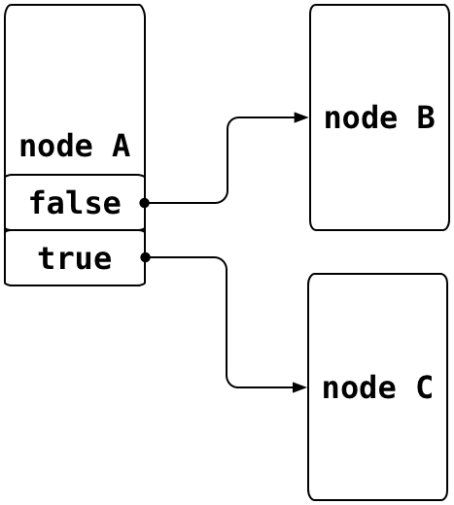
Each sandbox also has a set of **ports** associated with it. The ports are the points where network traffic essentially flows into and out of the sandbox. The way into an AFI sandbox is via an input port and the connection from a sandbox to the outside world is via an output port.

Forwarding path elements are called **nodes**. Few examples of AFI nodes are counters, discards, trees, flow tables and conditionals. All AFI nodes have a token assigned to them that uniquely identifies them within a sandbox. Connections from one node to another are made by referencing the token for the downstream node. In the example below, node A will only refer to node B via node B's token.



Nodes can have as many references to other nodes as they need.

A simple conditional if-else node would look like this:

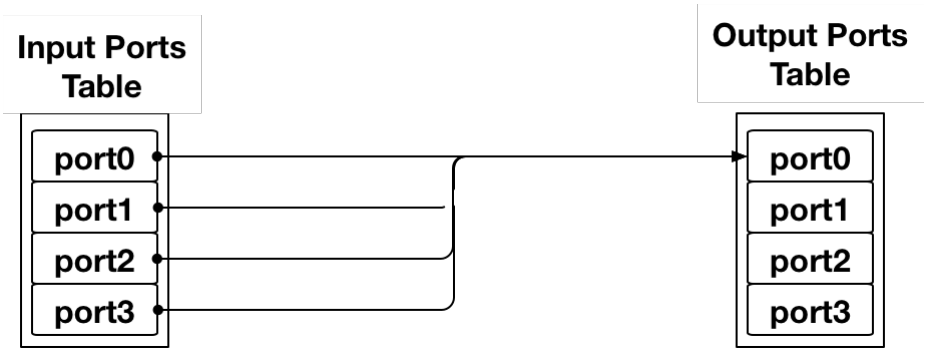


Clients create a path and insert nodes into it, to describe a forwarding graph (or path) between the input and output ports. Forwarding paths are often a collection of dynamic lookup containers, loosely connected together by whatever features are defined for a particular input or output interface.

Simple Sandbox Examples

1. Patch Sandbox

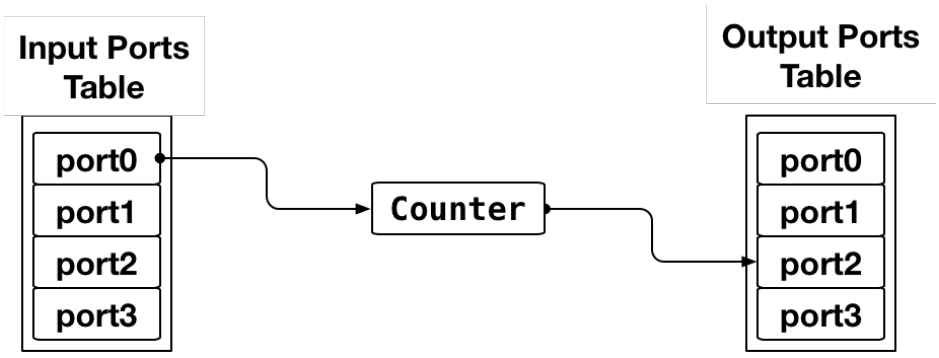
A very simple sandbox would look like



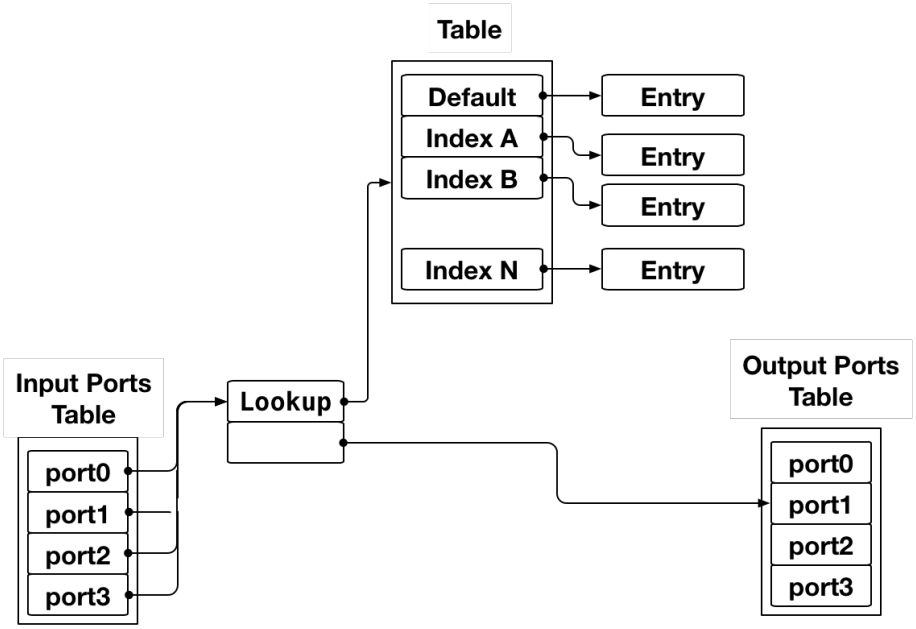
This is essentially a “patch” sandbox. All the four input ports connect to one output port. It provides a basic static function (all traffic from the input ports are directed to one output port) but there’s no dynamic element in the sandbox at all.

2. A Sandbox path with counter

Following example shows a simple sandbox where an input port connected to an output port with the counter in between.

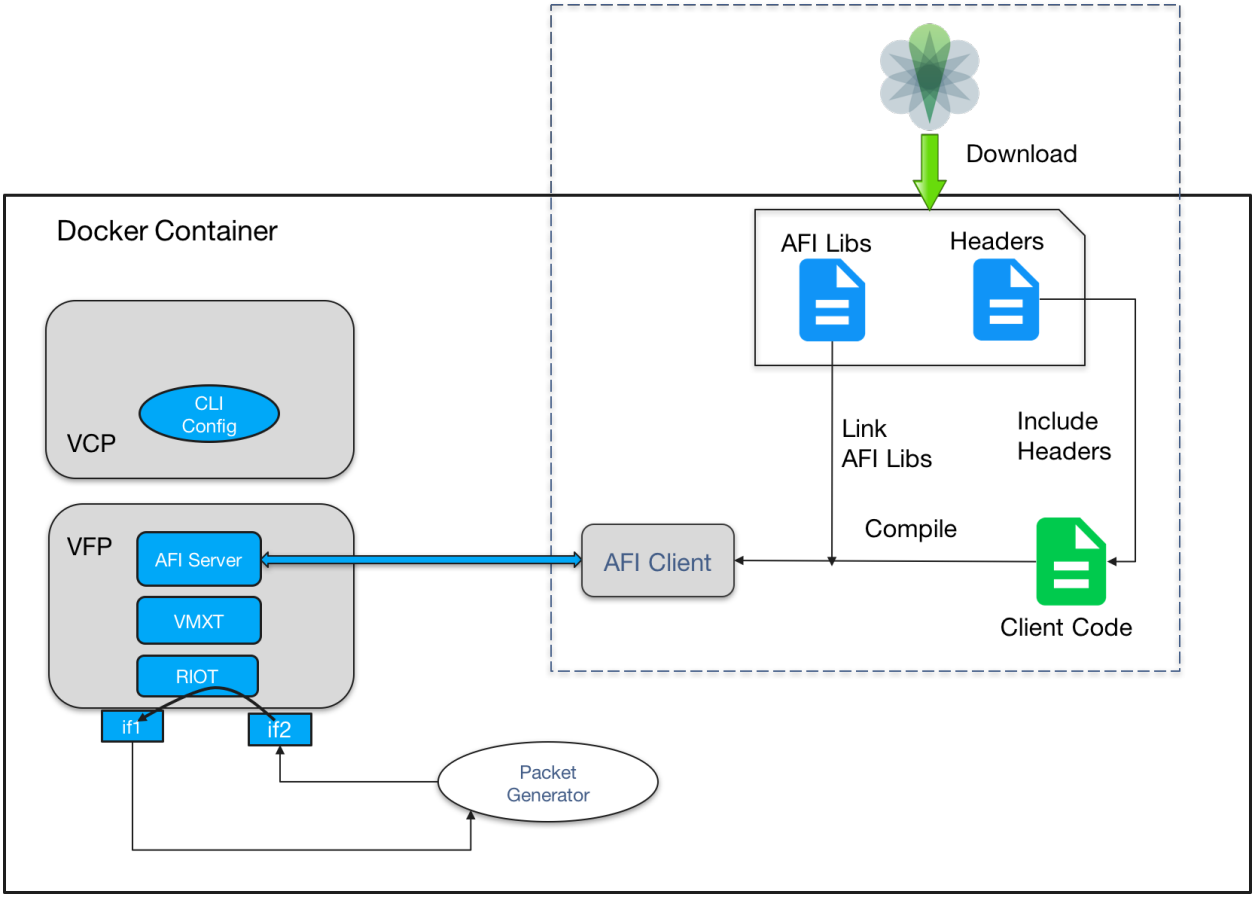


3. Sandbox with Lookup



In this example, all the input ports are connected to a list that executes a lookup in a table (which in turn, executes whatever is bound to the entry that matches the lookup) and then exits the sandbox through the first output port. This arrangement is very common in a sandbox to change a packet or metadata field based on the lookup of another value (such as an input interface or nexthop index).

3. AFI Client Development Environment and Workflow



Tarball containing AFI libraries and header is available for download from Juniper support/download page. AFI example clients can be found on AFI GitHub repository. Development and execution environment (using VMX running inside a Docker container) is provided.

AFI client developer will download VMX image published on Juniper Networks website, build a Docker container. Dockerfile to build Docker container is also provided on AFI GitHub repository. Detailed instructions are provided in the README.md of GIT repository for AFI.

Hardware requirement

Processor	Any x86 processor (Intel or AMD) with VT-d capability
Number of Cores	4 (1 for VCP and 3 for VFP)
Memory	Minimum: 8 GB (2 GB for VCP, 6 GB for VFP) Additional 2 GB recommended for host OS
Storage	Local or NAS. Minimum 200 GB is recommended.
Operating system	Ubuntu 14.04 LTS Linux 3.13.0-32-generic

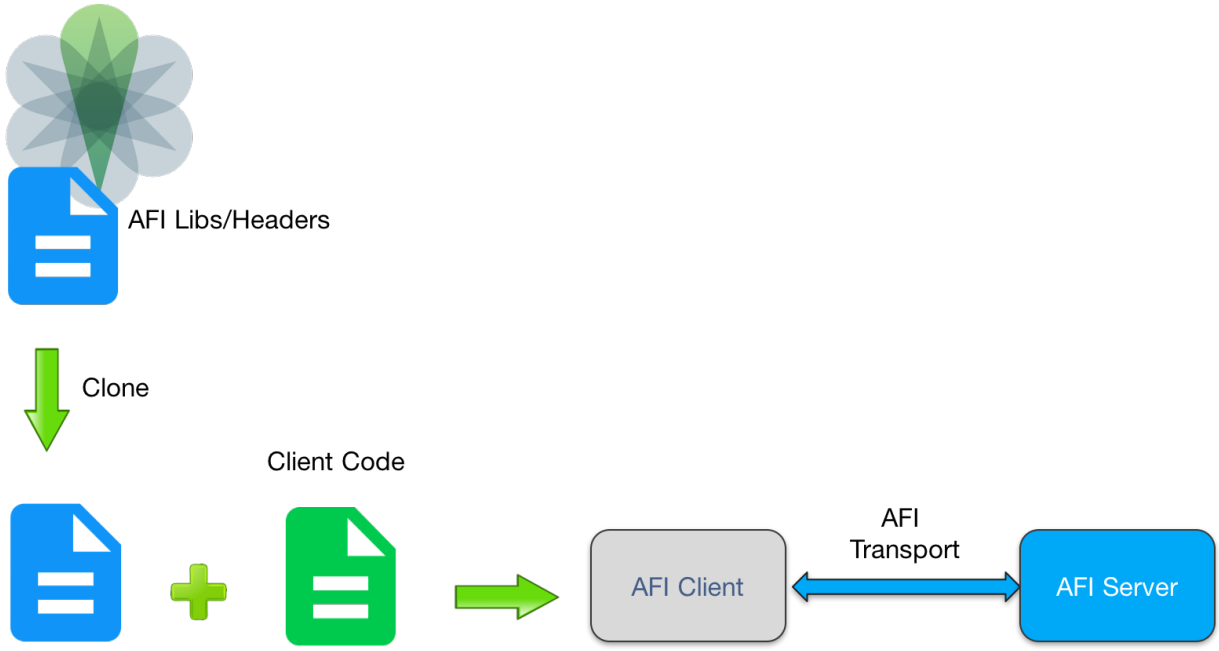
Steps to build and run Docker container with VMX image

Step 1	Pull AFI GIT repository.
Step 2	Go to **`afi/tools/docker`** directory
Step 3	Download the VMX image from Juniper's website.
Step 4	Build Docker container using Dockerfile/s present in afi/tools/docker directory
Step 5	Run Docker container

After these steps user will have a Docker container UP with VMX running inside it and AFI server running inside VFP VM of VMX.

AFI Client Development Workflow

AFI libs and header files will be hosted on GIT server. Third party developer will clone the AFI repository. AFI client will be written using AFI APIs (include header files). During compilation, the AFI client code will be linked with provided AFI libs to generate AFI client executable.



In the Docker host of the Docker container (provided as part of this RLI) the environment to develop and run AFI client will be available. This is provided so that AFI clients can be developed and validated easily.

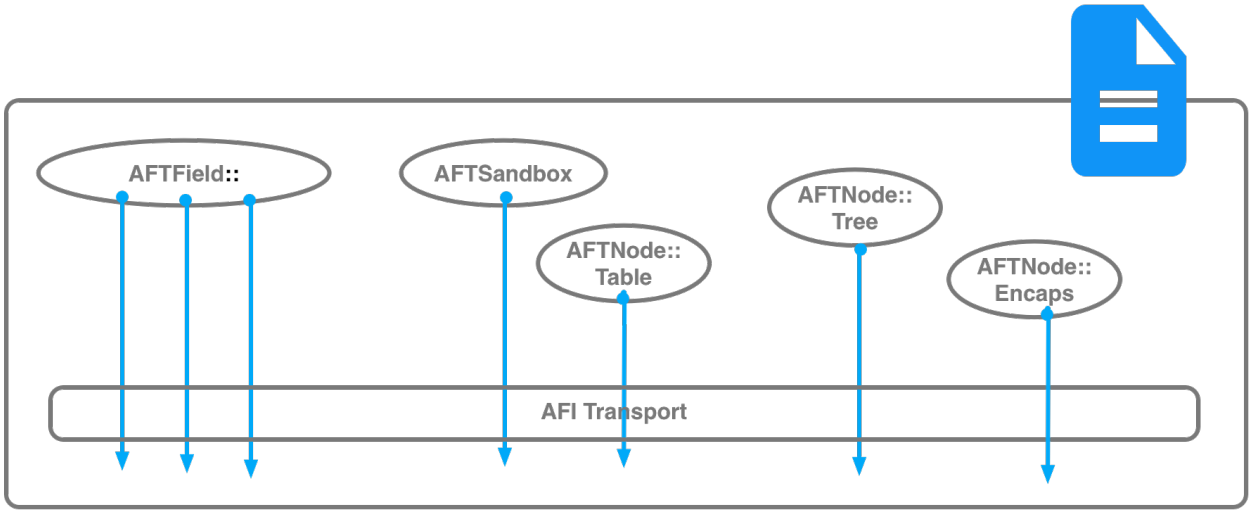
AFI GIT Repository Directory Structure

```
-- afi
    |-- README.md
    |-- tools
    |   `-- docker                  Files to build AFI Docker
    |-- docs
    |   `-- doxygen                 Doxygen documentation
    `-- example-clients             Reference AFI clients
        `-- afi-client
```

4. AFI API Interface

The AFI libraries define a set of C++ classes that, when used to create and connect objects, allow the underlying graph of nodes to be realized. The nodes themselves reflect the basic operations, not higher level primitives. In AFI, we call a collection of nodes a "path". The AFI library also provides support for name resolution of nodes, field definitions (for describing metadata and packet fields), dynamic table entries, encapsulation/de-encapsulation templates and other useful primitives for packet forwarding. Please note AFI does not provide native AFI APIs for other languages such C and Python.

Doxygen documentation for AFI APIs can be found in the AFI GIT repository at location '**`docs/doxygen/`**'. Example AFI clients are present at location '**`example-clients/`**' in the AFI GIT repository.



1. Sandbox Operations

Classes for accessing and managing AFI sandboxes

Class	AFI Release	Brief Description
`AftSandbox`	1.0	AftSandbox class provides interface for accessing and managing state (AFI nodes and entries).
`AftInsert`	1.0	Transactional insert used for inserting sandbox state. To add nodes/entries into sandbox path, AFI client creates insert context and first pushes the nodes/entries to the insert context. Sending the insert context to sandbox results in all these nodes/entries to be added to the sandbox path.
`AftRemove`	1.0	Transactional insert used for removing sandbox state. AFI client uses remove context to delete nodes/entries from sandbox path. AFI client pushes the nodes/entries, to be removed to the remove context. Sending the remove context to sandbox results in all these nodes/entries to be deleted from the sandbox path.

2. Nodes

Classes that define the core AFI forwarding nodes.

Class	AFI Release	Brief Description
AftNode	1.0	Base class for all AFI nodes. All AFI nodes are derived from a common super class (AftNode).
AftDeleteNode	1.1*	Class used to delete the referenced node token.

3. Basic Operations Nodes

Class	AFI Release	Brief Description
AftDiscard	1.0	Discard. When a discard node is added to the sandbox path and packet hits that node it results in the counter getting discarded.
AftContinue	1.1*	Continuation operation.
AftIndirect	1.1*	Indirect reference to another node.
AftMatch	1.1*	Simple two-way conditional. Using AftMatch “if() match” and “if()else match” conditionals can be added to sandbox path.
AftSwitch	1.1*	Static multi-part conditional. AftSwitch provides interface to add standard switch() like operation to sandbox path. All of the case values are provided when the AftSwitch is created.

4. Lookup Nodes

Class	AFI Release	Brief Description
AftTree	1.0	Variable-length key lookup tree container.
AftTable	1.0	AftTable provides support for scalar table container.
AftLookup	1.0	Key lookup class. Lookup node is used to perform a lookup in a lookup cointainer (e.g. Tree, Table) using a field or set of fields.

5. Counters and Policers

Class	AFI Release	Brief Description
AftCounter	1.0	When a counter node is added to the sandbox path and packet passes through that node it results in the counter getting incremented.
AftPolicer	1.1*	When a policer node is added to the sandbox it results in policing of the traffic passing through that node at specified policing parameters.

6. Load Balancing Node

Class	AFI Release	Brief Description
AftLoadBalance	1.1*	AftLoadBalance provides interface to create simple load balancer in the sandbox path. The load balancer is dynamic and uses an AftEntry for each leg of the load balancer table. The fields, to be used to generate the hash index into the table, are provided while creating load balancer.

7. List Nodes

Class	AFI Release	Brief Description
AftList	1.0	Simple non-dynamic list. The simple list node allows a sequence of nodes to be listed by their token values. This allows sequential operations to be defined.
AftIndexedList	1.1*	Indexed list is more complicated (than AftList) and dynamic list container.

8. Managing of Entry and Exit Points of Sandbox

Class	AFI Release	Brief Description
AftInputPort	1.0	The port objects come in two flavors - input and output. The way into an AFI sandbox is via an input port. AftInputPort class provides the interface to manage input port of sandbox
AftOutputPort	1.0	Connection from an AFI path to the outside world is via an output port. AftOutputPort class provides the interface to manage output port of sandbox.
AftSandboxLink	1.1*	Node to allow output ports to jump to another AftSandbox.

9. Other Nodes

Class	AFI Release	Brief Description
AftReceive	1.1*	Host receive class.
AftWrite	1.1*	Metadata field write class.
AftVariable	1.1*	Sandbox variable class.
AftParameters	1.1*	Optional dictionary of parameters for a node or an entry.

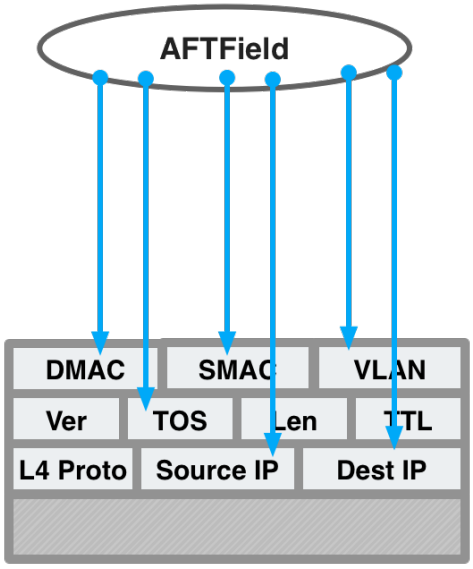
10. Entries

Classes that define the core AFI forwarding entries in lookup nodes

Class	AFI Release	Brief Description
`AftEntry`	1.0	Base class for all AFI entries. The AftEntry class is the other core superclass (other than AftNode) that is used widely throughout the AFI. Every entry in a route table, index table, flow table or even a indexed list is represented by an object class based on AftEntry.
`AftDeleteEntry`	1.0	Class used to delete the referenced entry.

11. Fields and Keys

Classes used to define packet fields and associated field/data tuples expressed as keys



Class	AFI Release	Brief Description
`AftField`	1.0	Simple class for describing packet (and other types) of fields.
`AftFieldEntry`	1.0	Class used to describe an individual field entry.
`AftFieldTable`	1.1	Table that holds all the known and legal fields for a sandbox.
`AftKey`	1.0	Tuple class for associating AftField and AftData values.

12. Data

Classes of basic data encapsulations that allow abstraction and re-use across the AFI interface.

Class	AFI Release	Brief Description
AftData	1.0	Base class for all AFI data classes.
AftDataInt	1.0	Data class for all integers.
AftDataPrefix	1.0	Data class for byte-array prefixes.
AftDataIP4Addr	1.0	Data class for IPv4 addresses.
AftDataIP6Addr	1.1	Data class for IPv6 addresses.
AftDataEtherAddr	1.0	Data class for Ethernet MAC addresses.
AftDataLabel	1.0	Data class for MPLS labels.

13. Encap/Decap

Classes used to manage data path packet encapsulation/de-capsulation.

Class	AFI Release	Brief Description
AftDecap	1.0	Packet de-encapsulation class.
AftEncap	1.0	Packet encapsulation class.
AftEncapTable	1.1*	Interface for managing encapsulation entries.
AftEncapEntry	1.1*	Individual entry in the encapsulation table.

14. Hostpath

Managing packets sent or received on hostpath interface

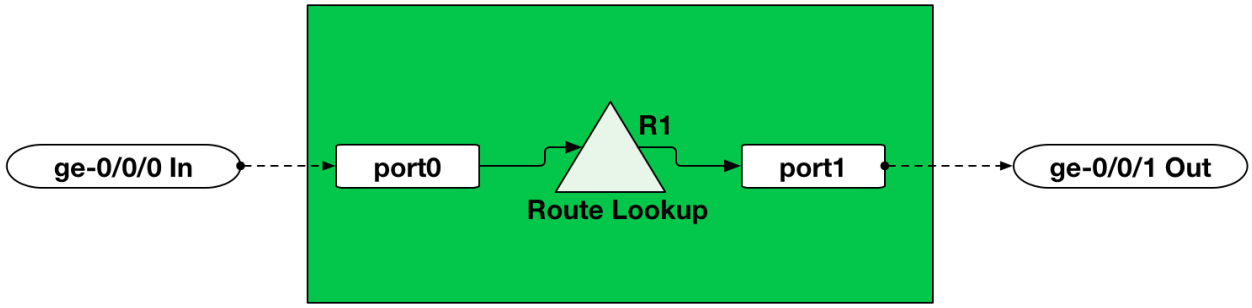
Class	AFI Release	Brief Description
AftPacket	1.0	AftPacket class provides interface for constructing the packet to be sent on hostpath connection and for parsing the packet received on the hostpath connection.

* Proposed release.

4.1.1 Sandbox Use cases

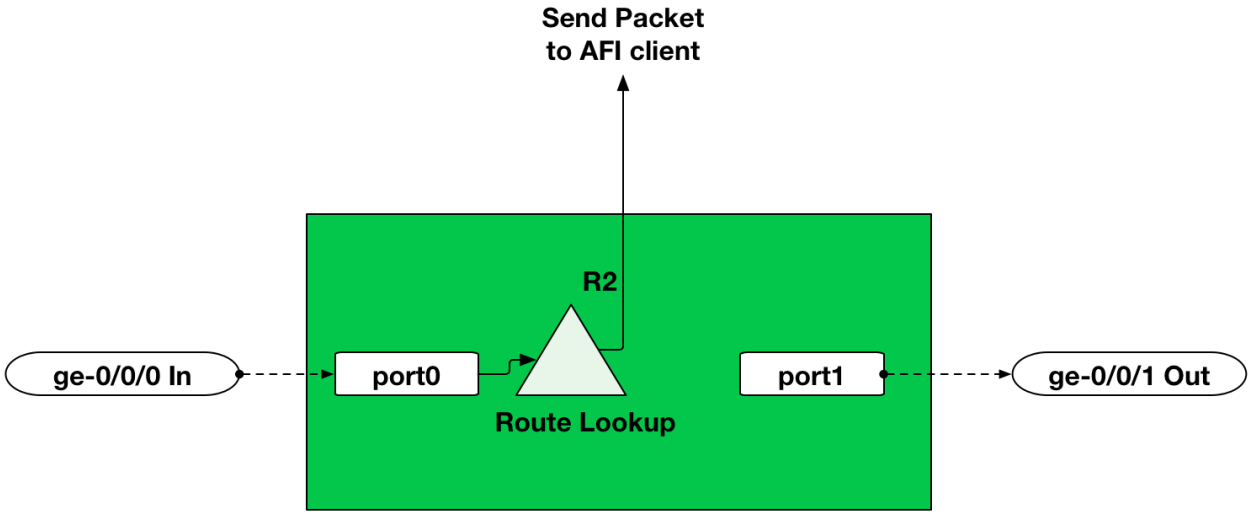
Few typical example sandbox use cases which can be realized using AFI.

1. Routing

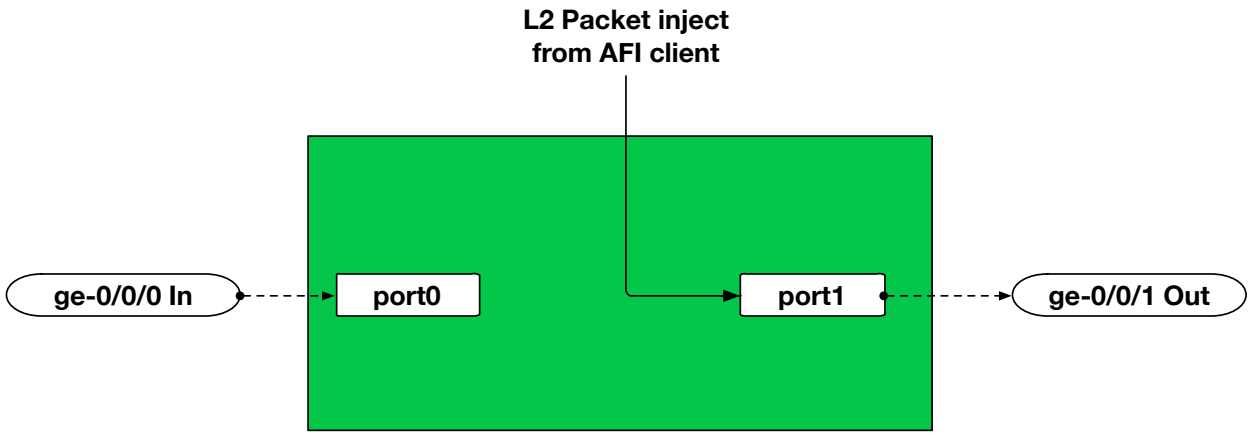


2. Packet punt to AFI client

AFI client packets hitting a route are sent to AFI client.



3. Layer 2 packet inject from AFI client



4.1.2 Example Client

Following example connects one input port to an output port with the counter in between.

```
//
// afi/example-clients/first-afi-client
//
// This example client code will make following sandbox topology
//
//               Sandbox: green
//        +------------------------------+
//        |                              |
//        |            list              |
//     p0 o------->+----------+          |
//  Input |        | counter  |          |
//   port |        |          |          |
//        |        +----------+          |
//        |        |       ---|--------->o p1
//        |        |          |          | Output Port
//        |        +----------+          |
//        |                              |
//        +------------------------------+
//
//

// The first thing an AFI client needs to do is initialize transport
// connection to server. The AftTransportPtr is a smart pointer to the Afi
// path context. As a rule, the Afi library uses C++ STL smart pointers
// for the persistent state that it manages in order to avoid the
// potential for memory issues.
AftTransportPtr transport = AfiTransport::create("128.0.0.16:50051");

// To open a sandbox, AFI client simply calls the transport open()
// method with the name of the sandbox AFI client wants to attach to.
transport->alloc("trio",   // Name of the engine
                 "green",  // Sandbox name
                  2,       // Number of input ports
                  2);      // Number of output ports

AftSandboxPtr sandbox;
transport->open("green",   // Name of sandbox transport controls
                sandbox); //

//
// Allocate an insert context
//
AftInsertPtr  insert = AftInsert::create(sandbox);

// This creates a simple counter with initial byte/packet count
// as 0/0

AftNodePtr counter =  AftCounter::create(0, 0, false);

// Add it to insert context
insert->push(counter);
```

```
//
// Get the token for output port "1"
//
AftNodeToken outputPortToken;
AftIndex outputPortIndex = 1;
sandbox->outputPortByIndex(outputPortIndex, outputPortToken);

//
// Build a list to describe the flow of lookup for this sandbox.
// For every packet, a counter will be increment and then packet
// exits via the outputPort.
//
AftNodePtr list = AftList::create({counter->nodeToken(), outputPortToken});
insert->push(list);

//
// Send all our nodes to the sandbox
//
sandbox->send(insert);

//
// Connect all the input port 0 to the list we just created
//
AftIndex inputPortIndex = 0;
sandbox->setInputPortByIndex(inputPortIndex, list->nodeToken());
```

This above example code illustrates a few fundamental AFI techniques. We tend to sequence sets of forwarding path operations using lists of nodes (which, in turn, describe the steps to take). In this case, we have a list that reference the counter we want to use and then punts the packet to the output port.

AFI client always creates and installs nodes before linking them into the path (i.e. like most forwarding structures, Aft paths are built in reverse). That's why the last operation to be performed is the setInputPortByIndex () call to connect the input port to the simple list.

5. CLI Configuration

Sandbox is created through Junos CLI configuration. When a sandbox is configured in Junos CLI, memory is reserved for sandbox in forwarding ASIC where the forwarding node graph will reside. Mappings of physical IFDs to ports are also provided in the sandbox configuration.

Statements to configure a forwarding sandbox will be at the [edit forwarding-options] hierarchy level. Following is the template of CLI configuration to create forwarding sandbox.

```
forwarding-options {
+    forwarding-sandbox <sandbox-name> {
+        size small/medium/large;
+        port <portname1> {
+            interface <interface>;
+        }
+        port <portname2> {
+            interface <interface>;
+        }
+        ...
+        ...
+    }
}
```

In this cli configuration <sandbox-name> is the string for sandbox name. “size” statement specifies the size (small/medium/large) of forwarding ASIC memory to be reserved for this forwarding sandbox. Name of the port specified after “port” statement can be any string. Interfaces, to which a sandbox port maps to, are IFDs.

Sandbox Configuration CLI Help

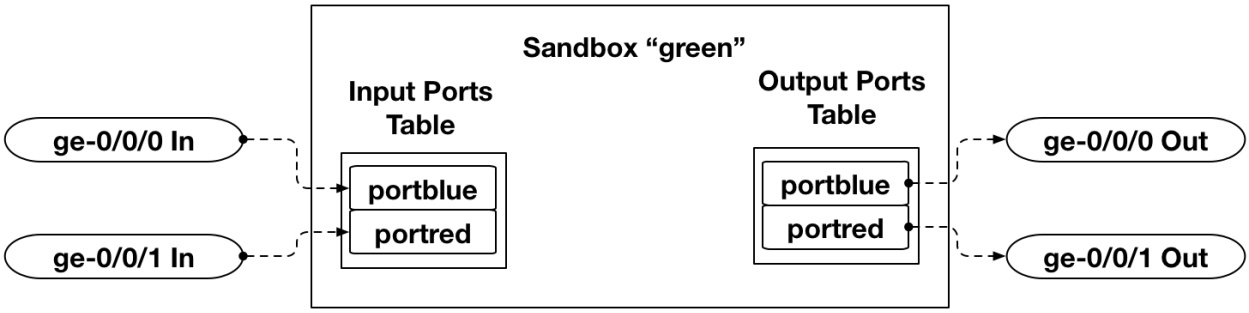
```
# set forwarding-options ?
> forwarding-sandbox              Create forwarding sandbox

# set forwarding-options forwarding-sandbox ?
  <forwarding-sandbox-name>       Forwarding sandbox name

# set forwarding-options forwarding-sandbox red ?
> size                            Size of the sandbox
> port                            Sandbox port

# set forwarding-options forwarding-sandbox red size ?
> small                           Sandbox size small
> medium                          Sandbox size medium
> large                           Sandbox size large

# set forwarding-options forwarding-sandbox port ?
  <port-name>                     Name of sandbox port

# set forwarding-options forwarding-sandbox port portblue ?
> interface                       Interface to which the port is mapped
```

```
# set forwarding-options forwarding-sandbox port portblue interface ?
> <interface>                     Interface to which the port is mapped
```

Sample configuration

Following Junos CLI configuration creates sandbox named “green” with two ports “portblue” (which is mapped to ge-0/0/0) and “portred” (which is mapped to ge-0/0/1”).

```
forwarding-options {
+    forwarding-sandbox green {
+        size small;
+        port portblue {
+            interface ge-0/0/0;
+        }
+        port portred {
+            interface ge-0/0/1;
+        }
+
+    }
}
```



6. Glossary

AFI	Advanced Forwarding Interface
Junos	Juniper's network operating system for advanced routing, switching, and security. Junos OS is the common operating system that runs across Juniper's routing, switching, and security devices.
GitHub	GitHub (https://github.com/) is a web-based Git repository hosting service.
PFE	Packet Forwarding Engine
VCP	Virtual Control Place
VFP	Virtual Forwarding Plane
VM	Virtual Machine
CLI	Command Line Interface

7. Links

- AFI Introduction (this document)
  https://github.com/Juniper/AFI/blob/master/docs/AFI.pdf
- GitHub AFI Repository (example clients and Docker scripts)
  https://github.com/Juniper/AFI
- Doxygen Documentation
  https://juniper.github.io/AFI/doxygen/html/index.html
- AFI Libs/Headers and 'AFI VMX' Download
  http://www.juniper.net/support/downloads/?p=vmxeval#sw